Práctica 1. Algoritmos devoradores

Francisco Javier Molina Rojas
javier.molinarojas@alum.uca.es
Teléfono: 722528757
NIF: 45386606Q

16 de diciembre de 2022

1. Describa a continuación la función diseñada para otorgar un determinado valor a cada una de las celdas del terreno de batalla para el caso del centro de extracción de minerales.

   La función que evalúa las celdas para colocar el centro de extracción se basa en puntuar su proximidad con el centro del mapa. Las celdas mas céntricas (aquellas cuya distancia hacia el centro es menor), serán la que mayor puntuación tengan. Esto lo hacemos con el objetivo de que, a la hora de colocar las defensas, poder colocar el mayor número de estas.

   Esto lo conseguimos al usar la función ya proporcionada en el "FAQ" de la asignatura "cellCenterToPosition", la cual nos proporciona la posición central de la casilla que le pasemos.

   Con la casilla dada, la casilla mas céntrica del mapa (con fila en la mitad del máximo de la fila y columna en la mitad del máximo de columna), y la función "_distance" (función que calcula la distancia entre dos posiciones céntricas), devolveremos 1000 entre la distancia de las posiciones (entre mas distancia, menos puntuación y lo mismo a la inversa)

Funcion cellValueExtrationCenter

```
float cellValueExtrationCenter(int row, int col, bool** freeCells
,float cellWidth , float cellHeight,int nCellsWidth, int nCellsHeight)
//Para poder evaluar cada celda en el caso de querer
//construir el centro de Extraccion y las defensas debemos intentar
//crearlo lo mas cercano al centro con el objetivo
//de que la posicion disponible mas centrica sea el centro de
//extraccion para poder construir el numero maximo de defensas alrededor suya
{
        if(freeCells[row][col])// Comprobamos que la celda que hemos seleccionado esta libre
        {
                //Debemos tener en cuenta que las casillas mas centricas sera las
                //que mas se acerquen a la posicion formada por la celda situada en
                //nCellsWidth/2(la mitad del ancho) en el caso de la row y a
                //nCellsHeight/2(la mitad de la altutra) en el caso de la col.
                //Para poder medir la distancias entre la celda que se nos pasa
                //y la celda central conseguiremos sus posisciones reales
                //gracias a la funcion proporcionada cellCenterToPosition
                // y gracias a la funcion \_distance podemos obtener la distancia entre ambos.
                //Por ultimo, para poder tener una ponderacion devolveremos el
                // siguiente dato: 1000/\_distance(Centro,celdaActual) debido a que
                //cuanto mas cercana sea la celda proporcionada, menor sera la
                //distancia entre ellas, por consecuente, mayor sera su puntuacion.
                Vector3 Centro = cellCenterToPosition(nCellsWidth/2,
                nCellsHeight/2,cellWidth,cellHeight);
                Vector3 celdaActual = cellCenterToPosition(row,col,
                cellWidth,cellHeight);
                return (float) 1000/_distance(Centro,celdaActual);
        }
        return -1; //No esta libre
}
```

2. Diseñe una función de factibilidad explicita y descríbala a continuación.

   La función de factibilidad comprobará si una celda es factible para colocar la defensa o no. Para ello, tendremos que comprobar que no este fuera de rango, que no choque con un obstáculo o defensa ya colocada y que no salga del mapa.

Para que la casilla no este fuera de rango, no tiene que ser menor que 0 tanto en fila como en columna y no tiene que ser mayor o igual que los extremos del tablero.

Para que no choque con los obstáculos o defensas, la distancia entre el centro de la celda candidata y la celda del obstáculo/defensa ya colocada no debe ser menor que la suma de los radios de la defensa a colocar y el obstáculo/defensa ya colocada.

Para que no salga la defensa del mapa tenemos que comprobar que su radio no viola los limites del mapa. Para ello si la posición de la fila o la posición de la columna menos el radio de la defensa es menor que 0 sobrepasa alguno de los extremos inferiores. Mientras que si la posición de la fila o la posición de la columna mas el radio de la defensa es mayor que la anchura o la altura del mapa, entonces sobrepasa alguno de los extremos superiores.

Función de factibilidad

```
bool EsFactible(int row, int col,List<Defense*>::iterator currentdefense
        , int nCellsWidth, int nCellsHeight, float mapWidth, float mapHeight,
        List<Object*> obstacles, List<Defense*> defenses,int numDefensasColocadas)
{
        //definimos una variable factible, la predefinimos con true,
        //y si encontramos que no es factible la pondremos a false
        bool factible = true;

        //si la row y col de la celda pasada esta dentro de rango
        if(row >= 0 && row < nCellsWidth && col >= 0 && col < nCellsHeight)
    {
                //definicon de variables necesarias para la verificacion de la factibilidad
                //obtenemos la posicion de la celda
                Vector3 pos = cellCenterToPosition(row,col,mapWidth/nCellsWidth
                ,mapHeight/nCellsHeight);
                //definimos un iterador para la lista de obstulos inicializado con la primera posicion
                List<Object*>::iterator poit = obstacles.begin();
                //definimos un iterador para la lista de defensas inicializado con la primera posicion
                List<Defense*>::iterator pdit = defenses.begin();

                //Comprobamos que los obstaculos no chocan con nuestra defensa
                //mientras que haya obstaculos por comprobar y factible sea verdadero
                while(poit != obstacles.end() && factible)
                {
                        //si la suma de los radios de la defensa a colocar y el radio del obstaculo es mayor que
                        //la distancia desde el centro de la posicion
                        //de la celda hasta el centro de la posicion del obstaculo
                        if(((*currentdefense)->radio + (*poit)->radio) >
                        _distance(pos,(*poit)->position))
                                factible = false; //entonces la defensa no es colocable ya que estaria
                                //intersectando el radio del ocbstaculo (factible es falso)
                        poit++; // pasamos al siguiente obstaculo
                }

                //Comprobamos que las defensas YA COLOCADAS no chocan con nuestra defensa
                //mientras que haya defensas por comprobar y factible sea verdadero
                while(pdit != defenses.end() && factible)
                {
                        //si la defensa que iteramos es la que vamos a colocar o
                        //numero de defensas colocadas es 0, no hay defensa que pueda colisionar
                        if(pdit == currentdefense || numDefensasColocadas == 0)
                        {
                                pdit++; //pasamos a la siguiente defensa
                                continue; //terminamos la iteracion actual en el bucle
                        }
                        else
                        {
                                //si la suma del radios de la defensa a colocar
                                //y el radio de la defensa ya colocada es mayor que
                                //la distancia desde el centro de la posicion de la
                                //celda hasta el centro de la posicion defensa
                                if( ((*currentdefense)->radio + (*pdit)->radio)
                                > _distance(pos,(*pdit)->position))
                                        factible = false; //entonces la defensa no
                                        //es colocable ya que estaria
                                //intersectando el radio de la otra defensa (factible es falso)
                                pdit++; // pasamos a la siguiente defensa
                                numDefensasColocadas--; //hemos comprobado esta defensa,
                                //por lo que ya no la contamos
                        }
                }

                //si la resta de la posicion x/y de la celda
                //menos el radio de la defensa es menor que 0
                //o la suma de la posicion x/y de la celda
                //mas el radio de la defensa es mayor que
                //la anchura o altura del mapa entonces el radio de la defensa
                //ocupa una parte inexistente del borde del mapa
                if(pos.x - (*currentdefense)->radio < 0
                || pos.y - (*currentdefense)->radio < 0
                ||pos.x + (*currentdefense)->radio > mapWidth
                || pos.y + (*currentdefense)->radio > mapHeight)
                        factible = false; //entonces la defensa no es colocable ya que
                        //estaria violando los bordes del mapa (factible es falso)
        }
        else    //Si row y col esta fuera de rango
                factible = false; //entonces la defensa no es colocable (factible es falso)

        return factible; //devolvemos la factible
}
```

3. A partir de las funciones definidas en los ejercicios anteriores diseñe un algoritmo voraz que resuelva el

problema para el caso del centro de extracción de minerales. Incluya a continuación el código fuente relevante.

Voraz para el centro de extracción

```
void DEF_LIB_EXPORTED placeDefenses(bool** freeCells, int nCellsWidth,
                        int nCellsHeight, float mapWidth, float mapHeight
            , std::list<Object*> obstacles, std::list<Defense*> defenses) {

    float cellWidth = mapWidth / nCellsWidth; //Ancho de celda
    float cellHeight = mapHeight / nCellsHeight; //Altura de celda
    int numDefensasColocadas = 0; //Numero de defensas colocadas

        //Definicion de variables que usaremos mas adelante:
        //Para la lista de candidatos usaremos una lista la
        //cual almacenara las celdas de manera ordenada conforme al valor de la celda
        std::list<Celda> Celdas;
        //Definimos una celda auxiliar para poder seleccionar un candidato de la lista
        Celda auxiliar(0,0,0);
        //definimos un iterador de la lista de defensas para poder colocar las defensas
        List<Defense*>::iterator currentDefense = defenses.begin();

        //Inicializamos la lista de candidatos para colocar el centro de extraccion
        for(int i = 0 ; i < nCellsWidth ; i++)
        {
                for(int j = 0 ; j < nCellsHeight ; j++)
                {
                        //insertamos todas las celdas en la lista de candidatos. Para poder
                        //ordenarla luego usaremos el metodo sort
                        Celdas.push_front(Celda(i,j,cellValueExtrationCenter(i,j
                        ,freeCells,cellWidth,cellHeight,nCellsWidth,nCellsHeight)));
                }
        }
        Celdas.sort(); //Se realiza la ordenacion (al final de
        //las listas estan las mejores celdas)

        //Comenzaremos colocando el centro de extraccion de minerales
        //(primera posicion de defensas)
        //Definimos una variable que nos indicara true cuando
        //el centro de extraccion sea colocado
        bool CentroColocado = false;
        // Mientras que hayan candidatos(celdas)disponibles
        //y el centro no haya sido colocado
        while(!Celdas.empty() && !CentroColocado)
        {
                auxiliar = Celdas.back(); //Guardamos el mejor candidato posible
                Celdas.pop_back(); //lo eliminamos de la lista de candidatos
                //Si la celda elegida es factible
                if(EsFactible(auxiliar.row,auxiliar.col,currentDefense,nCellsWidth,
                nCellsHeight,mapWidth,mapHeight,
                obstacles,defenses,numDefensasColocadas))
                {
                        //le damos a la primera defensa (Centro de Extraccion)
                        //la posicion de la celda elegida
                        (*currentDefense)->position =
                        cellCenterToPosition(auxiliar.row,auxiliar.col,
                        cellWidth,cellHeight);
                        //El centro ha sido colocado
                        CentroColocado = true;
                        //numero de defensas es 1
                        numDefensasColocadas++;
                }
        }
```

4. Comente las características que lo identifican como perteneciente al esquema de los algoritmos voraces.

   Como podemos ver, el algoritmo pose una lista de candidatos que contiene las celdas junto a las puntuaciones de estas, una variable que funcionará como la función solución (Indicando si el centro ha sido colocado o no), una función de selección que esta representada como la extracción del ultimo elemento de la lista (ya que al estar ordenada de menor a mayor valor de celda, la celda con mejor valor estará al final), una función de factibilidad que comprobara si la celda es factible o no, una función objetivo que es la una celda y el objetivo la celda mas céntrica.

5. Describa a continuación la función diseñada para otorgar un determinado valor a cada una de las celdas del terreno de batalla para el caso del resto de defensas. Suponga que el valor otorgado a una celda no puede verse afectado por la colocación de una de estas defensas en el campo de batalla. Dicho de otra forma, no es posible modificar el valor otorgado a una celda una vez que se haya colocado una de estas defensas. Evidentemente, el valor de una celda sí que puede verse afectado por la ubicación del centro de extracción de minerales.

   La función que evalúa las celdas para colocar el resto de las defensas, va a ser bastante similar a la hecha para el centro de extracción. Los únicos cambios que haremos será:

   1. Pasar la lista de candidatos por referencia con el objetivo de poder asignar el nuevo valor a las celdas directamente.

   2. En vez de usar la distancia desde el centro de la celda hasta el centro de la celda más céntrica, usaremos el centro de la celda donde está colocado el centro de extracción.

Funcion cellValueDefenses

```
// Para poder evaluar las celdas en el caso de querer construir las defensas,
// haremos algo similar que con el centro de extraccion
//solo que ahora colocaremos las defensas lo mas cercano al centro de extraccion.
//Para ello modificaremos el valor de las celdas candidatas con el nuevo valor
void cellValueDefenses(std::list<Celda>& Celdas,
List<Defense*>::iterator CentroDeExtraccion, float cellWidth, float cellHeight)
{
        List<Celda>::iterator cit = Celdas.begin(); //definimos iterador
        while(cit != Celdas.end())//mientras haya celdas
        {
                //pasamos la celda actual a posicion
                Vector3 celdaActual =
                cellCenterToPosition(cit->row,cit->col,cellWidth,cellHeight);
                // el valor sera 100 entre la distancia de la celda actual y el centro de extraccion
                cit->valor =
                (float) 1000/_distance(celdaActual,(*CentroDeExtraccion)->position);
                //cambiamos de celda
                cit++;
        }
}
```

6. A partir de las funciones definidas en los ejercicios anteriores diseñe un algoritmo voraz que resuelva el problema global. Este algoritmo puede estar formado por uno o dos algoritmos voraces independientes, ejecutados uno a continuación del otro. Incluya a continuación el código fuente relevante que no haya incluido ya como respuesta al ejercicio 3.

Voraz total

```
void DEF_LIB_EXPORTED placeDefenses(bool** freeCells, int nCellsWidth,
                int nCellsHeight, float mapWidth, float mapHeight
        , std::list<Object*> obstacles, std::list<Defense*> defenses) {

    float cellWidth = mapWidth / nCellsWidth; //Ancho de celda
    float cellHeight = mapHeight / nCellsHeight; //Altura de celda
    int numDefensasColocadas = 0; //Numero de defensas colocadas

        //Definicion de variables que usaremos mas adelante:
        //Para la lista de candidatos usaremos una lista la
        //cual almacenara las celdas de manera ordenada conforme al valor de la celda
        std::list<Celda> Celdas;
        //Definimos una celda auxiliar para poder seleccionar un candidato de la lista
        Celda auxiliar(0,0,0);
        //definimos un iterador de la lista de defensas para poder colocar las defensas
        List<Defense*>::iterator currentDefense = defenses.begin();

        //Inicializamos la lista de candidatos para colocar el centro de extraccion
        for(int i = 0 ; i < nCellsWidth ; i++)
        {
                for(int j = 0 ; j < nCellsHeight ; j++)
                {
                        //insertamos todas las celdas en la lista de candidatos. Para poder
                        //ordenarla luego usaremos el metodo sort
                        Celdas.push_front(Celda(i,j,cellValueExtrationCenter(i,j
                        ,freeCells,cellWidth,cellHeight,nCellsWidth,nCellsHeight)));
                }
        }
        Celdas.sort(); //Se realiza la ordenacion (al final de
        //las listas estan las mejores celdas)

        //Comenzaremos colocando el centro de extraccion de minerales
        //(primera posicion de defensas)
        //Definimos una variable que nos indicara true cuando
        //el centro de extraccion sea colocado
        bool CentroColocado = false;
        // Mientras que hayan candidatos(celdas)disponibles
        //y el centro no haya sido colocado
        while(!Celdas.empty() && !CentroColocado)
        {
                auxiliar = Celdas.back(); //Guardamos el mejor candidato posible
                Celdas.pop_back(); //lo eliminamos de la lista de candidatos
                //Si la celda elegida es factible
                if(EsFactible(auxiliar.row,auxiliar.col,currentDefense,nCellsWidth,
                nCellsHeight,mapWidth,mapHeight,
                obstacles,defenses,numDefensasColocadas))
                {
                        //le damos a la primera defensa (Centro de Extraccion)
                        //la posicion de la celda elegida
                        (*currentDefense)->position =
                        cellCenterToPosition(auxiliar.row,auxiliar.col,
                        cellWidth,cellHeight);
                        //El centro ha sido colocado
                        CentroColocado = true;
                        //numero de defensas es 1
                        numDefensasColocadas++;
                }
        }

        //Actualizamos el valor de las celdas para que sea el mejor valor para las celdas
        cellValueDefenses(Celdas,currentDefense,cellWidth,cellHeight);
        //Se realiza la ordenacion (al final de las listas estan las mejores celdas
        Celdas.sort();

        //pasamos a la siguiente defensa
        currentDefense++;

        //Continuamos con el colocado de las defensas restantes
        //establecemos un numero maximo de intentos
        int maxAttemps = 1000;
        //Mientras que haya defensas por colocar y haya candidatos disponibles
        //y el numero de intentos sea superior a 0
    while(currentDefense != defenses.end() && !Celdas.empty() && maxAttemps > 0)
```

```
{
            //seleccionamos el mejor candidato disponible
            auxiliar = Celdas.back();
            //lo eliminamos de la lista de candidatos
            Celdas.pop_back();
            //Si la celda elegida es factible
    if(EsFactible(auxiliar.row, auxiliar.col, currentDefense, nCellsWidth,
    nCellsHeight, mapWidth, mapHeight, obstacles, defenses, numDefensasColocadas))
    {
                    //asignamos a la defensa la posicion de la celda elegida
            (*currentDefense)->position =
            cellCenterToPosition(auxiliar.row,
            auxiliar.col, cellWidth, cellHeight);
                    //pasamos a la siguiente defensa
                    currentDefense++;
                    //defensa colocada +1
                    numDefensasColocadas++;
    }
            //restamos 1 al numero de intentos
            maxAttemps--;
}
```

Todo el material incluido en esta memoria y en los ficheros asociados es de mi autoría o ha sido facilitado por los profesores de la asignatura. Haciendo entrega de este documento confirmo que he leído la normativa de la asignatura, incluido el punto que respecta al uso de material no original.